**February 3, 1998**

# TEST ASSERTIONS FOR DATE AND TIME FUNCTIONS

**Gary E. Fisher**
Computer Scientist
National Institute of Standards and Technology (NIST)

## CHANGES

Changes in this version:

- Editing of definitions.
- Addition of test dates to Tables 2 and 3.

Changes in 1/13/98 version:

- Corrections to definitions for a.m., cc, hh, leap year, p.m., ss, tt, week-of-year, and ww.
- Replaced cc with tt.
- Corrections to day-of-week(), DOW(), dow(), and decrDate().
- Corrections to Table 11.
- Removed the term "midday".
- Modified column subscripts and reference subscripts to include table number (e.g., column_4.2 refers to Table 4, column 2.)
- Added SQL Test Suite reference.

Changes in 7/23/97 version:

- Corrections to entries in tables 4, 5, and 15.
- Rename table 6.

Changes in 6/16/97 version:

- Addition of time zone change assumption.
- Addition of quarter, week-of-year, and ww definitions.

## PURPOSE

This specification defines assertions for use in testing date and time functions. A product that implements and passes applicable tests written using these assertions can be said to be compliant with this specification. The test assertions are based on generic functions that encompass functionality exhibited in various programming language, database, network, and operating system specifications and standards. The functions were developed using the common functionality from numerous sources and application requirements in general.

Date/time functions encompass formats for display and information interchange, date arithmetic, and sorting. Assertions are defined for each of these areas. The relationship of the number of test cases to assertions is not straightforward, i.e., an assertion may require one or more test cases to completely cover the constraints of the assertion. Further, an assertion is applicable only if a real implementation provides the specified functionality.

## SCOPE

Since there are an infinite number of variations on date and time processing within the set of applications that exist in the world today, it is impossible in one document such as this to develop test assertions and their respective tests for every possible algorithm and use of date and time functionality. Test assertions can be developed for a limited set of functionality.

This specification takes some of the common functionality found in many applications and standards, and describes it in the form of an application programming interface (API) consisting of functions, arguments, and results using a pseudo-syntax and semantics. The API is not complete since there are variations on each function and the number and types of arguments that could be used. Users, however, should be able to apply the described functionality in this document to the functionality described and used in other applications, and thereby develop a set of tests that can be used in various environments.

Possible uses of this specification include--

- the development of a reference implementation that could be used to test applications through communications channels or remote procedure calls; that is, a correct program based on these assertions would produce output that is used or interpreted by an application. The tested application would be judged on how well it accepted or manipulated the output from the reference implementation.
- the development of test data sets that could be processed by a test implementation. Output from the test implementation would be compared with standard results from this specification, and the test implementation output judged on how well it matched the standard results.
- the development of Request for Proposal (RFP) requirements and test plans. Bidders would submit applications, source code, test data, and test output that implement these tests. This information would be evaluated by source selection evaluators and become part of the RFP submission and review process. In addition, organizations would use this specification as input to the test planning process.

The assertions cover the following functions and computations:

- formatted display and information interchange of date and time
- localization of date and time formats
- conversion of date and time (military time, a.m./p.m., separators, and string/numeric conversion)
- date and time arithmetic
- sorting of date and time data elements

Certain assumptions apply to the use of these assertions. They are listed as follows:

- Insofar as these assertions are based on the Gregorian calendar system and the Latin-1 character set, the use of the assertions assumes that resultant answers based on different calendar systems or different character sets may be possible, but are not within the scope of this specification.
- Characters used in these assertions assume those in the Latin-1 set of Unicode and may be single- or multi-byte characters internally.
- No other assumptions are made about the internal storage characteristics of date and time

information.
- No assumptions are made about the source of date and time information, i.e., whether the information comes from a program, operating system, network, database, user, etc.
- If and when Daylight Savings Time (DST) comes into play, tests based on these assertions should be modified to take into account changes in dates and times arising from Daylight Savings Time in those instances where it may be appropriate and is not already included in the test assertions.
- Specific functions covering relative time changes between time zones (e.g., converting from time zone A to time zone B) are not provided. These are viewed as special cases of generic functions applied to specific applications and can be tested by combining requirements of component tests.
- Capabilities outside, and in addition to, the functions described here are not considered in the definition of assertions.
- Each complete set of date and time value assertions presented in the tables within this document should be used in tests to ascertain that results are consistent over a range of dates and times, and through specific trouble spots, such as boundary conditions that have been identified.

## DEFINITIONS

(NOTE: These definitions may be replaced by those compiled in the draft IEEE P2000.1, Standard Date and Time Definitions, a specification that is currently in ballot.)

**a.m.** ante meridiem; displayed, on a 12-hour clock, after times whose hours would be in the interval 00 to 11:59:59.99 on a 24-hour clock. (Noon is defined as 12 noon and midnight as 12 midnight, neither of which is a.m. nor p.m.)

**cc** a 2-digit representation of the thousands and hundreds part of a 4-digit year descriptor (e.g., cc of 1998 is 19.)

**century year** a year that is divisible by 100 with no remainder.

**day-of-week** a single digit descriptor representing the ordinal of each weekday valid over the interval 1 through 7, such that Monday is 1, Tuesday is 2, and so on, through Sunday which is 7.

**DD** a 2-digit day-of-month descriptor valid over the intervals defined in Table 1, column 1.2.

**DST** Daylight Savings Time; by Act of the U.S. Congress, the period from 2:00 a.m. on the first Sunday in April to 2:00 A.m. on the last Sunday in October during which 1 hour is added to the offset from UTC of the local standard time zone.

**hh** a 2-digit hour descriptor valid over the interval 00 to 23 on a 24-hour clock, such that midnight is 00 and noon is 12.

**JJJ** a 3-digit ordinal day descriptor valid over the interval 001 through 365, or 001 through 366 in a leap year, commencing with the first day of the year, January 1.

**leap year** a noncentury year that is divisible by 4 with no remainder; or a century year that is divisible by 400 with no remainder, such that the month of February is expanded by an intercalary day on February 29.

**localization** the translation or conversion of a date according to local custom or law, such as the insertion of separators between date components, or representation in a number base that differs from the decimal base, or conversion to a different character set.

**MM** a 2-digit month descriptor valid over the interval 01 through 12.

**mm** a 2-digit minute descriptor valid over the interval 00 to 59.

**MMM** a 3-character all-capitalized abbreviation of the names of months as defined in Table 7, column 7.2.

**mmm** a short-hand method of referencing the full names of months as defined in Table 7, column 7.3.

**ordinal date** a 7-digit date of the form YYYYJJJ, where YYYY is the ordinal of the year, and JJJ is the ordinal of the day-within-year.

**p.m.** post meridiem; displayed, on a 12-hour clock, after times whose hours would be in the interval 12 to 23:59:59.99 on a 24-hour clock. (Noon is defined as 12 noon and midnight as 12 midnight, neither of which is a.m. nor p.m.)

**quarter** a 1-digit ordinal representing the position of the quarter-year in which a specific date falls, such that all dates between January 1st and March 31st inclusive fall in quarter 1, April 1st and June 30th in quarter 2, July 1st and September 30th in quarter 3, and October 1st and December 31st in quarter 4. Alternately, quarter may refer to any 3-month period within a fiscal year, member year, or other context-dependent time boundary based on 12 month periods.

**ss** a 2-digit second descriptor valid normally over the interval 00 to 59, or 00 to 58 when a leap second is subtracted from the year, or 00 to 60 when a leap second is added to the year.

**tt** a 2-digit hundredths-of-a-second descriptor valid over the interval 00 to 99. This is a minimum specification of time precision, although specific applications may require finer precision.

**TZ** abbreviation for time zone.

**TZoffset** the number of hours difference between local time and the time at the prime meridian, such that local time in different time zones can be calculated by adding or subtracting the TZoffset to or from the local time in Greenwich, England (located on the prime meridian at longitude 0). The difference is positive if the local time is ahead of UTC, and negative if behind. For example, 12:00 noon Eastern Standard Time in the United States is 18:00Z-6:00, on a 24-hour clock, where Z (zulu) represents UTC at Greenwich, England. -6:00 is the TZoffset. During Eastern Daylight Savings Time, the TZoffset would be -5:00.

**UTC** Coordinated Universal Time; time standard set by international agreement on Temps Atomique Internationale (ATI); replaced Greenwich Mean Time (GMT) as the international time standard.

**valid date** any date within the interval October 15, 1582, the beginning of the Gregorian calendar, through December 31, 4082, (the Gregorian calendar is accurate within 1 day for approximately 2,500 years) and within the parameters of months and number of days per month used in this specification. (Character representations of these dates according to ANSI X3.30-1988 would appear as 15821015 and 40821231. Internal representation in a form other than character leads to limits in the interval that can be stored. For example, a 32-bit integer can store up to 136 years in seconds, but cannot store the complete interval of dates as described above. The same 32-bit integer can store 11.8 million years of dates when number-of-days is stored.)

**week-of-year** an ordinal number within the range 01 through 53 inclusive that describes the ordinal position of each week within a calendar year. The first week of the year is the first week in which a Thursday occurs (by international business convention). The first week on the calendar for 1999 is December 28, 1998, through January 3, 1999, but since the Thursday in that week occurs on December 31st, the first week of 1999 is January 4-10. The 52nd week of 1999 includes the dates December 27, 1999, through January 2, 2000.

**ww** a 2-digit week-of-year ordinal descriptor valid over the interval 01 through 53.

**YYYY** a 4-digit year ordinal descriptor valid over the interval 1582 through 4082.

## GENERIC FUNCTIONS

The following definitions of generic functions are used within this specification. They are specified using a C-like syntax, such that each function has the form,

(castType) optionalResultVar = pseudoFunction([optionalArg_1, optionalArg_2, ...])

The functions that are apparent in a particular application may not match the data types and functionality specified in this document, but the concept of the function will surely have a related use within the implementation under scrutiny. It is up to the user to make the shift and apply the concept of testing defined here to the particular implementation.

Each table of values is made up of columns that are numbered with subscripts. Following each generic function is a list of one or more assertions that references the table columns. For example, the assertion

assert(pseudoFunction(column_*t*.1) == TRUE)

defines a set of tests. The reference to column_*t*.1 denotes that values in column 1 of the associated table, *t*, are inserted for each specific test. All tests within a specific set of assertions must return true for a function to be considered conformant to this specification.

(logical) tf = validDate(YYYYMMDD)

tests date elements, such that true is returned for YYYY in the year interval 1582 to 4082 inclusive, MM in the month interval 1 to 12 inclusive, and DD in the day interval imposed by the mapping in table 1. Otherwise, false is returned.

Table 1. Month to Number-of-days Mapping

| Month (MM)$_{1.1}$ | Day-of-month (DD) Interval$_{1.2}$ |
|---|---|
| January | 01 through 31 |
| February<br>February in leap year | 01 through 28<br>01 through 29 |
| March | 01 through 31 |
| April | 01 through 30 |
| May | 01 through 31 |
| June | 01 through 30 |
| July | 01 through 31 |
| August | 01 through 31 |
| September | 01 through 30 |
| October | 01 through 31 |
| November | 01 through 30 |
| December | 01 through 31 |

assert(validDate(column_2.1) == column_2.2)

---

Table 2. Date Validation Examples

| Date$_{1.2}$ | Valid?$_{2.2}$ |
|---|---|
| 19990909 | TRUE |
| 19991231 | TRUE |
| 19999999 | FALSE |
| 20000101 | TRUE |
| 20000228 | TRUE |
| 20000229 | TRUE |
| 20000301 | TRUE |
| 19990228 | TRUE |
| 19990229 | FALSE |

(logical) tf = leapYear(YYYY)

tests a year, such that ((YYYY % 4 == 0) && (YYYY % 100 != 0)) || (YYYY % 400 == 0) yields true, otherwise, false. Table 3 contains the test values associated with this function. (% represents the modulus operator.)

assert(leapYear(column_3.1) == column_3.2)

---

Table 3. Leap Year Test Examples

| $\text{Date}_{3.1}$ | $\text{Leap Year?}_{3.2}$ |
|---|---|
| 1900 | FALSE |
| 1999 | FALSE |
| 2000 | TRUE |
| 2001 | FALSE |
| 2004 | TRUE |

(int) doy = dayOfYear(YYYYMMDD)

if given a valid date, returns the ordinal day-of-year of that date, such that the ordinal of the day within the year is in the interval 1 through 365 (366 in a leap year) starting with January 1 as ordinal day 1. Table 4 defines values for testing this function.

assert(dayOfYear(column_4.1) == column_4.2)

assert(dayOfYear(19990229) == ERROR)

NOTE: Since ERROR is not defined and the assert statement most likely cannot operate in this fashion, the concept that an ERROR code or other nonnormal action takes place is what is relevant. Users are encouraged to develop methods of testing ERROR conditions specifically for an individual implementation. Implementations may not signal an error, but the resulting answer shall not be interpreted by an implementation as a valid response.)

---

Table 4. Ordinal Day-of-year Examples

| $\text{Date}_{4.1}$ | Resulting $\text{Day-of-year}_{4.2}$ | Resulting Ordinal $\text{Date}_{4.3}$ |
|---|---|---|
| 19991231 | 365 | 1999365 |
| 20000101 | 1 | 2000001 |
| 20000228 | 59 | 2000059 |
| 20000229 | 60 | 2000060 |
| 20000301 | 61 | 2000061 |
| 19990228 | 59 | 1999059 |

(int) ord = ordinalDate(YYYYMMDD)

such that a single 7-digit number is returned in the form YYYYJJJ, where JJJ is defined by dayOfYear().

assert(ordinalDate(column_4.1) == column_4.3)

(int) YYYYMMDD = date()

returns system current date, such that elements of the date adhere to the constraints of validDate().

assert(validDate(date()) == TRUE)

NOTE: This assertion may have to be tested by changing the system date to various values. In some cases, assigning an invalid date may not be possible, but in those cases where is it possible, the test should prove this.

((long) hhmmss || (long) hhmmsstt) = time()

returns a system clock time, such that hh is in the interval 00 to 23 inclusive, mm is in the interval 00 to 59 inclusive, ss is in the interval 00 to 59 inclusive, and the optional tt is in the interval 00 to 99 inclusive. Fractional seconds may be displayed to more decimal places depending on precision requirements and system capabilities.

No assertions are defined. See the SQL standard test suite [SQL1996] for examples of how date/time precision may be tested.

((long$^*$) YYYYMMDDhhmmss || (long$^*$) YYYYMMDDhhmmsstt) = dateTime()

returns the current date and time group, such that the elements of the returned value adhere to valid date() and time() values. ($^*$Optionally, these values may be represented in some implementations as character strings.)

No assertions are defined. See related assertions on date() and time() for parallels to this function.

(char) dowName= dayOfWeek(YYYYMMDD)

returns the name of the day of the week corresponding to the given date, such that the day-of-week string is as defined in table 5.

assert(dayOfWeek(column_5.4) == column_5.2)

assert(dayOfWeek(19990229) == ERROR)

---

Table 5. Day-of-week Number, String Name, and Abbreviated String Name Values

| Day-of-week number$_{5.1}$ | Day-of-week string$_{5.2}$ | Abbreviated day-of-week string$_{5.3}$ | Test Date$_{5.4}$ |
|---|---|---|---|
| 1 | Monday | MON | 20010103 |
| 2 | Tuesday | TUE | 20000229 |
| 3 | Wednesday | WED | 19970326 |
| 4 | Thursday | THU | 20001207 |
| 5 | Friday | FRI | 19991231 |
| 6 | Saturday | SAT | 20000101 |
| 7 | Sunday | SUN | 20000102 |

(char) abbrDow = DOW(YYYYMMDD)

returns an abbreviated day-of-week name for the corresponding given date, such that the abbreviated day-of-week string is as defined in table 5. Optionally, abbreviated day-of-week strings may be displayed as lower case with first letter capitalized.

assert(DOW(column_5.4) == column_5.3)

assert(DOW(19990229 == ERROR)

(char) YYYY = extract(YYYYMMDD, "YYYY")

returns a 4-digit year string extracted from a given date. Table 6 contains the appropriate test values.

assert(extract(column_6.1, "YYYY") == column_6.2)

assert(extract(19990229, "YYYY") == ERROR)

---

Table 6. Date Extraction Values

| Date$_{6.1}$ | Extracted Year$_{6.2}$ | Extracted Month$_{6.3}$ | Extracted Day-of-month$_{6.4}$ |
|---|---|---|---|
| 19991231 | 1999 | 12 | 31 |
| 20000101 | 2000 | 1 | 1 |

(char) MM = extract(YYYYMMDD, "MM")

returns a 2-digit month string extracted from a given date.

assert(extract(column_6.1, "MM") == column_6.3)

assert(extract(19990229, "MM") == ERROR)

(char) DD = extract(YYYYMMDD, "DD")

returns a 2-digit day-of-month string extracted from a given date.

assert(extract(column_6.1, "DD") == column_6.4)

assert(extract(19990229, "DD") == ERROR)

(char) monthName = month(monthNumber)

returns the name of the corresponding monthNumber as defined in table 7.

assert(month(column_7.1) == column_7.2)

(char) monName = MON(monthNumber)

returns the abbreviated month name of the corresponding monthNumber as defined in table 7. Optionally, abbreviated month strings may be displayed as lower case with first letter capitalized.

assert(MON(column_7.1) == column_7.3)

---

Table 7. Month Number, String Name, and Abbreviated String Name

| Month Number$_{7.1}$ | Month String$_{7.2}$ | Abbreviated Month String$_{7.3}$ |
|---|---|---|
| 1 | January | JAN |
| 2 | February | FEB |
| 3 | March | MAR |
| 4 | April | APR |
| 5 | May | MAY |
| 6 | June | JUN |
| 7 | July | JUL |
| 8 | August | AUG |
| 9 | September | SEP |
| 10 | October | OCT |
| 11 | November | NOV |
| 12 | December | DEC |

(char) dateFmt = localDateFormat()

returns a string containing a print format for local date display. Localized options for formatted date display shall include one or more of the elements in table 8.

assert(localDateFormat() == oneOf(enum(column_8.2))

NOTE: The oneOf() function is used to denote that a valid response is any one or more of the values in the specified column.

---

Table 8. Local Date Format Examples

| Given Date$_{8.1}$ | Date Format String$_{8.2}$ | Date Displayed$_{8.3}$ |
|---|---|---|
| 19991231 | YYYY/MM/DD | 1999/12/31 |
| 19991231 | YYYY-MM-DD | 1999-12-31 |
| 19991231 | YYYY.MM.DD | 1999.12.31 |
| 19991231 | MM/DD/YYYY | 12/31/1999 |
| 19991231 | MM-DD-YYYY | 12-31-1999 |
| 19991231 | MM.DD.YYYY | 12.31.1999 |
| 19991231 | MM DD YYYY | 12 31 1999 |
| 19991231 | MMM DD, YYYY | DEC 31, 1999 |
| 19991231 | DD MMM YYYY | 31 DEC 1999 |
| 19991231 | (none) | 19991231 |
| 20000101 | (none) | 20000101 |
| 19991231 | mmm dd, yyyy | December 31, 1999 |
| 20000101 | MMM DD, YYYY | JAN 1, 2000 |
| 19991231 | dd mmm yyyy | 31 December 1999 |
| 20000101 | DD MMM YYYY | 01 JAN 2000 |

(char) timeFmt = localTimeFormat()

returns a string containing a print format for local time display. Localized options for formatted time display shall include one or more of the elements in table 9.

assert(localTimeFormat() == oneOf(enum(column_9.2)))

---

Table 9. Local Time Formats

| Given Time$_{9.1}$ | Time Format String$_{9.2}$ | Time Displayed$_{9.3}$ |
|---|---|---|
| 11595999 | hh:mm:ss.tt | 11:59:59.99 |
| 11595999 | hh:mm:ss | 11:59:59 |
| 11595999 | hh:mm | 11:59 |
| 11595999 | hhmmss.tt | 115959.99 |

(long) countOfDays = daysBetween(startDate, endDate)

returns the number of days between a starting date and an ending date, such that the ending date is greater than or equal to the starting date. Table 10 defines assertion values.

assert(daysBetween(column_10.1, column_10.2) == column_10.3)

assert(column_10.1 <= column_10.2)

assert(daysBetween(column_10.1, 19990229) == ERROR)

assert(daysBetween(19990229, column_10.2) == ERROR)

---

Table 10. Count of Days Between Two Dates

| Start Date$_{10.1}$ | Ending Date$_{10.2}$ | Resulting Count$_{10.3}$ |
|---|---|---|
| 19991231 | 20000228 | 59 |
| 19991231 | 20000301 | 61 |
| 19981231 | 19990301 | 60 |
| 19951231 | 19960228 | 59 |
| 19951231 | 19960301 | 61 |
| 19991231 | 20000101 | 1 |
| 19991231 | 19991231 | 0 |
| 19990228 | 19990229 | ERROR |

(int) resultDate = incrDate(startDate, countOfDays)

returns a new valid date which is greater than the start date by the number of days in countOfDays. Table 11 defines the assertions appropriate to this function.

assert(incrDate(column_11.1, column_11.2) == column_11.3)

assert(incrDate(19990229, column_11.2) == ERROR)

(int) resultDate = decrDate(startDate, countOfDays)

returns a new valid date which is less than the start date by the number of days in countOfDays. Table 11 defines the assertion values appropriate to this function.

assert(decrDate(column_11.3, column_11.2) == column_11.1)

assert(decrDate(19990229, column_11.2) == ERROR)

---

Table 11. Increment and Decrement a Date by a Count Examples

| Starting Date$_{11.1}$ | Increment/ (Decrement)$_{11.2}$ | Resulting Date$_{11.3}$ |
|---|---|---|
| 19991231 | 59 | 20000228 |
| 19991231 | 61 | 20000301 |
| 19981231 | 60 | 19990301 |
| 19951231 | 59 | 19960228 |
| 19951231 | 61 | 19960301 |
| 19991231 | 1 | 20000101 |
| 19991231 | 0 | 19991231 |

(int) dowNum = DOW(YYYYMMDD)

is specified as a periodic function based on the day-of-week definition in table 2. For any calendar date specified, a corresponding valid day-of-week number shall result as defined by the assertion values in table 12.

assert(dayOfWeekNum(column_12.1) == column_12.2)

Table 12. Day of Week Number Examples

| Date$_{12.1}$ | Resulting Day-of-week$_{12.2}$ |
|---|---|
| 19991231 | 6 |
| 20000101 | 7 |
| 20000228 | 2 |
| 20000229 | 3 |
| 20000301 | 4 |
| 19990228 | 1 |
| 19990229 | ERROR |

(int) ordDate = ordinalDate(YYYYMMDD)

returns the ordinal date in the form YYYYJJJ for the corresponding calendar date given. Converting calendar date to ordinal date is tested using the values in table 13.

assert(ordinalDate(column_13.1) == column_13.2)

assert(ordinalDate(19990229 == ERROR)

Table 13. Calendar Date to Ordinal Date Conversion Examples

| Calendar Date$_{13.1}$ | Resulting Ordinal Date$_{13.2}$ |
|---|---|
| 19991231 | 1999365 |
| 20000101 | 2000001 |
| 20000228 | 2000059 |
| 20000229 | 2000060 |
| 20000301 | 2000061 |
| 19990228 | 1999059 |
| 19990229 | ERROR* |

(int) calDate = calendarDate(YYYYJJJ)

returns a calendar date of the form YYYYMMDD for the corresponding ordinal date given. Converting ordinal date to calendar date is tested using the values in Table 13. (*A test for ERROR to calendarDate() cannot be constructed.)

assert(calendarDate(column_13.2) == column_13.1)

assert(calendarDate(19990229) == ERROR)

(char) time12 = clock12(hhmmss || hhmmsstt)

returns a time based on a 12-hour clock of the form hhmmsstt a.m. or p.m., such that a.m. is appended if the resulting time is between midnight and noon and p.m. is appended if the resulting time is between noon and midnight. Noon is defined as 12:00 noon and midnight is defined as 12:00 midnight. Table 14 defines the values to test this function. Daylight Savings Time (DST) assertions are included in order to test the move to and from DST according to U.S. law.

assert(clock12(column_14.1) == column_14.2)

assert(clock12(240001.00 == ERROR)

Table 14. 24-hour to 12-hour Clock Conversion Examples

| 24-hour Time$_{14.1}$ | Resulting 12-hour Time$_{14.2}$ |
|---|---|
| 120000.00 | 120000.00 noon |
| 120001.00 | 120001.00 p.m. |
| 000001.00 | 000001.00 a.m. |
| 010000.00 | 010000.00 a.m. |
| 240001.00 | ERROR |
| 19970406 013001.00 | 19970406 013001.00 a.m. |
| 19970406 020000.00 | 19970406 030000.0 a.m. DST |
| 19971026 015959.99 DST | 19971026 015959.99 a.m. DST |
| 19971026 020000.01 DST | 19971026 010000.01 a.m. |
| 19971025 235959.00 DST | 19971025 115959.00 p.m. DST |
| 19971026 235959.00 | 19971026 115959.00 p.m. |
| 19971026 010100.00 | 19971026 010100.00 a.m. |
| 000000.00 | 120000.00 midnight |
| 115959.99 | 115959.99 a.m. |

(char) tz = timeZone()

returns a string representing the local time zone setting.

No assertions defined for this function.

(float) tzoffset = timeZoneOffset()

returns a floating point number in the interval -13 to +13 inclusive representing the number of hours offset between the time in the local time zone and the UTC time, such that the current local time plus the tzoffset is equal to the UTC time. Table 15 contains samples of values to test this function.

assert(time() + timeZoneOffset() == UTC)

Table 15. Time Zone Offset Examples

| Current Local Time$_{15.1}$ | TZ Offset$_{15.2}$ | UTC$_{15.3}$ |
|---|---|---|
| 1300 EST | 5 | 1800 |
| 1300 EDST | 4 | 1700* |
| 1300 | 9.5 | 0330 |
| 0100 | -3 | 2200** |
| 2300 | 2 | 0100** |

(* The UTC is always expressed in terms of a base time since it is not attached to a local time zone, i.e., UTC does not have Daylight Savings Time. In this case, UTC has not changed, but the current local time is offset one hour less than standard time for the time zone.)

(**If the addition of the time zone offset results in an invalid time, the time is from the previous day for negative results, and from the following day for positive results. Add the invalid time to 2400 to get the actual local time, e.g., 0100 - 3 = -0200, which translates to 2400 - 0200 = 2200).

**SORTING DATE AND TIME DATA ELEMENTS**

Given a list of data elements containing date and time strings, a sorted ascending list of these data elements shall result in a lexicographical sequence of date and time elements that ascend in character value from first to last. When two elements differ only in their length, the longer element shall sort in sequence after the shorter one, so long as comparable date and time elements are included in each string (i.e., a string consisting only of a time will not sort correctly with another that contains only a date.) Table 16 illustrates the starting unsorted list and the resulting sorted list.

Table 16. Unsorted and Sorted Date/time Strings

| Unsorted List$_{16.1}$ | Resulting Sorted List$_{16.2}$ |
|---|---|
| 19991231 | 19970331120000.00 |
| 20000228240000.00 | 19991231 |
| 19970331120000.00 | 19991231235959.99 |
| 20000228 | 1999365 |
| 1999365 | 20000228 |
| 19991231235959.99 | 20000228240000 |

**OTHER FUNCTIONS**

Of particular interest are functions that use date and time parameters to compute other results, such as financial calculations, i.e., payment amortization, net present value, etc. Most database implementations contain definitions for supporting these functions, and most take time periods into account. Tests for demonstrating the accuracy of these functions should also be developed.

Functions that provide results based on indirect computations concerning date and time information also include time stamps on transaction processing, database backup and restore functions, network and system access and security based on time periods, and many others that are not included in this specification. All of these should be tested in addition to functions that are directly affected by date and time. A good treatise on testing these types of functions and additional date/time processing errors that are made in many applications can be found in [GTE1996].

## OTHER DATE/TIME STANDARDS AND REFERENCES

Much of the information used in the preparation of this document comes from existing standards and references on date/time. Some of the references used in this document are included here for further information.

[BOR1994] "Borland dBase for Windows 5.0 Language Reference", Borland International, Inc., Scotts Valley, CA, 1994, pp. 1042.

[GTE1996] "Proposed Criteria for `Century Compliance'", GTE Government Systems Corporation, PA96014, Waltham, MA, 1996, http://www.mitre.org/research/cots/GTE_CRITERIA.htm, March 25, 1997.

[IEEE1992] IEEE Std 2003.1-1992, "Test Methods for Measuring Conformance to POSIX, Part 1 C System Interfaces", The Institute of Electrical and Electronics Engineers, Inc., New York, 1992, pp. 442.

[ISO8601] ISO 8601:1988-06-14 and Technical Corrigendum 1 1991-05-01, "Data Elements and Interchange Formats -- Information Interchange -- Representation of Dates and Times", American National Standards Institute (ANSI), New York, 1988, pp. 14.

[Kuhn1997] Markus Kuhn, "A Summary of the International Standard Date and Time Notation", http://www.ft.uni-erlangen.de/~mskuhn/iso-time.html#zone, March 25, 1997.

[USC15] United States Code, Title 15, Chapter 6, Section 260a, "Advancement of time or changeover dates".

[L81996] Draft ANSI Standard for "Representation of Date for Information Interchange", X3L8/96-050, ASC NCITS L8, December 9, 1996,.

[L81997] Draft ANSI Standard for "Representation of Time for Information Interchange", X3L8/97-008gV2, ASC NCITS L8, March 23, 1997.

[SQL1996] NISTIR 5998, "Users Guide for the SQL Test Suite, Version 6," December 1996, National Institute of Standards and Technology, U.S. Department of Commerce.